ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ»
(Н И У « Б е л Г У »)

ИНСТИТУТ ИНЖЕНЕРНЫХ ТЕХНОЛОГИЙ И ЕСТЕСТВЕННЫХ НАУК

КАФЕДРА МАТЕМАТИЧЕСКОГО И
ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИНФОРМАЦИОННЫХ СИСТЕМ

РАЗРАБОТКА И РЕАЛИЗАЦИЯ ЭФФЕКТИВНОГО АЛГОРИТМА ОПТИМИЗАЦИИ ТРАФИКА ДЛЯ ПРОГРАММНО-КОНФИГУРИРУЕМЫХ СЕТЕЙ

Выпускная квалификационная работа
обучающегося по направлению подготовки 02.04.01 Математика и Компьютерные науки
очной формы обучения, группы 07001631
Денисова Ильи Андреевича

Научный руководитель
к.т.н., доцент
Чашин Ю.Г.

Рецензент
к.т.н., доцент
Урсол Д.В.

БЕЛГОРОД 2018

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В XXI веке постоянный рост подключенных устройств к сети «Интернет» повлёк за собой экспоненциальный рост общемирового трафика. Этот резкий скачок стимулировал развитие новых более быстрых и инновационных компьютерных сетей под названием «Программно-конфигурируемые сети». Новое поколение коммуникационных сетей имело важное преимущество в виде увеличенной пропускной способности каждого передающего устройства, однако главным минусом данных сетей стало время создания нового соединения, она увеличилась в более чем два раза. [4]

Проблемы вызваны сложностью построения маршрутов на крупных сегментах подконтрольных сетей, в которых количество подключенных передающих устройств больше 300 штук.

Теория построения кратчайших путей на графах имеет огромное влияние на современный мир. Благодаря своему широкому спектру предоставляемых возможностей, она в последнее время интенсивно развивается: вместе с улучшением уже разработанных методов изобретаются принципиально новые.

Поиск путей является важной задачей, которую используют в различных областях и сферах. Правильно рассчитанный маршрут, может сэкономить средства на единичную или групповую доставку, большой объём данных будет быстрее доставлен пользователю, искусственный интеллект в компьютерных играх покажет более интересное поведение и маршрут для вашего автомобиля будет построен эффективно с точки зрения затраченного времени на путь.

Проблемой поиском путей начали заниматься ещё в XIX веке. Тогда появилась первая задача по поиску путей – задача Коммивояжёра. Сегодня эта задача является классической для комбинаторики и поиска маршрутов.

Начиная с 1969 года компьютерные сети начали развиваться. Изначально в самой первой сети под название ARPANET было соединено 4 суперкомпьютера того времени. Благодаря этой сети была опробована маршрутизация пакетов с помощью протокола IP, который используется и по сей день. Начиная с 1990 года появилась всемирная паутина, которая соединила между собой самые удалённые уголки мира.[2]

Начиная с 1990 года сложность компьютерных сетей растёт в геометрической прогрессии становясь всё более объёмными и запутанными. В современных компьютерных сетях поиск пути осуществляется на два шага вперёд, потому что полный поиск пути может занять огромное количество времени, однако такой «быстрый» поиск не позволяет всегда построить самый оптимальный и быстрый маршрут [17]. Из-за возросшей сложности современных сетей им на смену разрабатывают программно-конфигурируемые сети суть, которых заключается в том, чтобы вынести логику построения маршрутов из передающих устройств и оставить устройствам только саму передачу данных. Для того, чтобы это реализовать необходимо использовать быстрый и эффективный алгоритм построения маршрута, который будет учитывать нагрузку на каждое отдельное устройство, чтобы добиться максимальной производительности сети.

Актуальность разработки эффективного алгоритма оптимизации трафика для программно-конфигурируемых сетей обусловлена нежеланием части игроков ИТ рынка переходить на новое поколение сетей в связи с большими задержками при создании соединений. Поэтому сейчас очень важно разработать быстрый и эффективный алгоритм для построения маршрута с помощью которого задержки при создании нового соединения существенно снизятся.

Исходя из вышесказанного, целью выпускной квалификационной работы является исследование алгоритмов построения маршрутов, разработка и оптимизация их скорости выполнения для программно-конфигурируемых сетей.

В ходе работы были поставлены следующие задачи:

1) Анализ проблем построения маршрутов в программно-конфигурируемых сетях;

2) Выбор инструментальных средств для реализации алгоритма, эмуляции и тестирования реализованного алгоритма на базе программно-конфигурируемых сетей;

3) Разработка и реализация алгоритма, тестирование на эмулированном сегменте программно-конфигурируемой сети;

4) Апробация алгоритма;

В 1 главе «Анализ проблем построения маршрутов в программно-конфигурируемых сетях» будет произведён анализ программно-конфигурируемых сетей, их слабые места и недостатки построения маршрутов внутри сложных сегментов сети. Также будет произведено обоснование выбора среды разработки, эмулятора программно-конфигурируемой сети для отладки и сетевой операционной системы.

Во 2 главе «Проектирование и реализация» будет произведено проектирование и реализация алгоритма.

В 3 главе «Апробация алгоритма» приведены данные о результатах проведённых испытаний относительно производительности разработанного алгоритма и сравнения с его аналогами.

В заключении сделан вывод о степени достижения поставленных целей и задач.

Данная работа состоит из 70 страниц, 7 рисунков, 22 листингов, 40 литературных источников и 2 приложений.

1. Анализ проблем построения маршрутов в программно-конфигурируемых сетях

1.1. Актуальность проблемы построения маршрутов в программно-конфигурируемых сетей

Проблема задержки перед созданием соединения для програмнно-конфигурируемых сетей является самым главным фактором, который останавливает их распространение по всему миру. Связано это с тем, что сервер, который называется программно-конфигурируемый контроллер, производит прокладку маршрута в сложных сегментах сети, в которых количество узловых устройств в текущий момент может доходить до 1000 штук. Цена данных прокладок намного выше чем в классических сетях, в них среднее время прокладки маршрута в сложном сегменте сети будет занимать порядка 14-20 микросекунд, а в программно-конфигурируемых 200 и более микросекунд. Для решения этой проблемы необходима разработка нового алгоритма.[15]

1.2. Общие сведения о программно-конфигурируемых сетях

Программно-конфигурируемые сети — это новое поколение компьютерных сетей, в которых ключевым отличием является вынос логики маршрутизации за пределы устройства на отдельный выделенный сервер. Появление этого поколения связано с экспоненциальным ростом трафика и падением эффективности текущих компьютерных сетей. [40]

В ПКС уровни управления сетью и передачи данных разделяются с помощью вынесения функций управления (коммутаторами, маршрутизаторами и т. п.) в приложения, работающие на выделенном сервере, называемом контроллере. Первые концепции таких сетей были

сформулированы специалистами университетов Стэнфорда и Беркли еще в 2006 году, а произведённые ими исследования получили одобрение не только в научных кругах, но и в сфере бизнеса. Эти идеи тепло встретили такие производители сетевого оборудования как, как Cisco, HP и прочие. [20] В марте 2011 года был основан консорциум Open Networking Foundation (ONF). Учредителями данного сообщества являются ряд крупных и влиятельных компаний в сфере IT. Этими компаниями являлись: Verizon, Yahoo, Microsoft, Google, Deutsche Telekom, и Facebook. Состав ONF быстро пополнился и другими не менее известными компаниями, таких как Marvell, Citrix, IBM, NEC, Brocade, Oracle, HP, Dell, Ericsson, и ряд других. Самая первая практическая реализация ПКС была предложена компанией Nicira, которая недавно стало частью VMware.

Интерес ИТ-компаний к ПКС вызван тем, что такая технология позволяет повысить эффективность сетевого оборудования на 25–30%, снизить на 30% затраты на эксплуатацию сетей, позволяет превратить управление сетями из творческого проектирования в инженерию, повысить защищённость сети и позволить пользователям самостоятельно с помощью программ создавать новые сервисы и оперативно загружать, и использовать их на сетевом оборудовании.

В большинстве наработок и исследований ключевые моменты в ПКС связаны с развиваемой в США программой Global Environment for Network Innovations (GENI) исследования будущего Интернета, включающая в себя порядка 40 ведущих университетов США. Деятельность объединенного центра Стэнфорда и Беркли, производящего различные изучения, исследования, эксперименты и наработки в области Internet2; а также с Седьмой рамочной программой исследований Европейского Союза Ofelia и проектом FEDERICA. [38]

Основные концепции ПКС:

- Вынесение процесса управления данных из передающего устройства на выделенный сервер, а процессы передачи данных оставить на передающих устройствах;
- Унифицированный и независимый интерфейс между уровнем управления и уровнем передачи данных;
- логически централизованное управление сетью, осуществляемое с помощью контроллера с установленной сетевой операционной системой и реализованными поверх сетевыми приложениями;
- виртуализация физических ресурсов сети;

Главная проблема текущих компьютерных сетей заключается в том, что приблизительно 20-25% передающее устройство тратит на прокладку маршрута по сегменту сети. Концепция программно-конфигурируемых сетей заключается в том, чтобы убрать логику построения маршрутов из устройства передачи данных и оставить ему только передачу данных. Новый тип сетей предлагает размещать логику передачи данных на специальных серверах, называемых программно-конфигурируемыми контроллерами, на которых происходит вычисление маршрута в рамках сегмента сети. Данные сети в первую очередь нацелены на центры обработки данных, ведь именно они отвечают за большой медиаконтент предоставляемый на различных ресурсах. [22]

За счёт централизованности управления сегментом сети данная сеть предлагает ряд серьёзных преимуществ:

- Снижение стоимости оборудования, за счёт упрощения оборудования;
- Более детальное управление сетью;
- Очень низкая вероятность несанкционированного доступа к ресурсам сети;

- Возможность разработки или доработки существующих инструментов для программно-конфигурируемых сетей для управления ресурсами сетей и потоками данных;

В рамках развития концепции данной технологии был разработан специальный протокол передачи данных OpenFlow. Этот протокол базируется на концепции управления обработки потоков данных. Принцип работы протокола прост, если коммутатор получает данные для определённого потока, он смотрит в специальные таблицы потоков, именуемые как flow tables, в которых указаны действия что необходимо совершить в случае получения данного потока, если он является новым, то тогда коммутатор запрашивает информацию у контроллера, в последствии контроллер производит создание нового маршрута и заносит его в таблицы потоков устройств, участвующих в процессе передачи данных данного потока.

Главное проблемой данного подхода является огромное количество информации о продвижении пакетов, называемой forwarding state explosion. Из-за этого нагрузка на контроллер является огромной и огромные сегменты сети, включающие в себя более 1000 передающих устройств, являются непосильной ношей с точки зрения задержек и стабильности передающей инфраструктуры. [19]

1.2.1. Архитектура программно-конфигурируемых сетей

В архитектуре ПКС(software-defined network) имеется три уровня управления сетью (Рис. 1.1.):

1. инфраструктурный уровень, предоставляющий набор сетевых устройств таких как, коммутаторов и каналов передачи данных;
2. уровень управления, включающий в себя сетевую операционную систему, которая обеспечивает приложениям сетевые сервисы и программный интерфейс для управления сетевыми устройствами и сетью;

3. уровень сетевых приложений для гибкого и эффективного управления сетью;

Первый пункт позволяет быстро изменять и модифицировать инфраструктуру сети без полного перестроения всего сегмента сети.

Второй пункт решает проблему недостаточно эффективного использования ресурсов передающих устройств для передачи данных.

Третий пункт даёт широкие возможности для управления доступом к сети, управлению передачей данных, модификации генерации маршрутов и прочих других возможностей. [5]

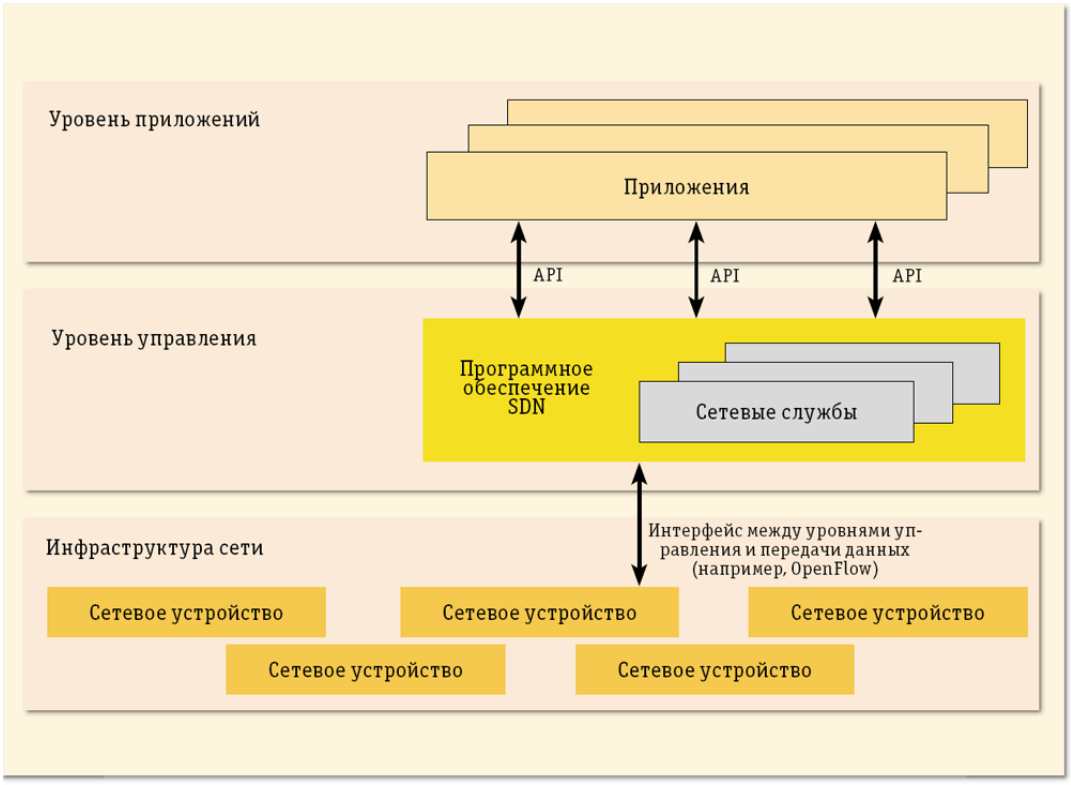

Рис. 1.1. Архитектура программно-конфигурируемых сетей

Наиболее многообещающим и быстро развивающимся эталоном для ПКС является протокол OpenFlow. Это открытый протокол взаимодействия, разработанный специально для ПКС, в котором описываются требования, предъявляемые к коммутатору, поддерживающему протокол OpenFlow для удаленного управления. [3]

С помощью современных маршрутизаторов повсеместно выполняются две главные задачи: передача данных — передача пакета из входного порта на определенный выходной порт и управление данными — обработка пакета и принятие решения о том, куда его передавать дальше, на основе текущей загруженности маршрутизатора и других факторов. [7] Это отражает уровень передачи данных, на котором объединены средства передачи, различные линии связи, каналообразующее оборудование, маршрутизаторы, коммутаторы, и уровня управления состояниями средств передачи данных (Рис. 1.2.). Развитие маршрутизаторов по сегодняшний день двигалось в направлении объединения этих уровней, однако со ставкой на передачу, включая в себя различные аппаратные ускорения, совершенствование ПО и внедрение новых функциональных возможностей для увеличения скорости обработки каждого пакета, в то же время уровень управления оставался достаточно примитивным и опирался на непростые распределенные алгоритмы маршрутизации и сложные инструкции по настройке и конфигурированию сети. В силу сложности программного обеспечения маршрутизаторов, в рамках которых разработан уровень управления, компании разработчики устройств закрывали исходный код. [22]

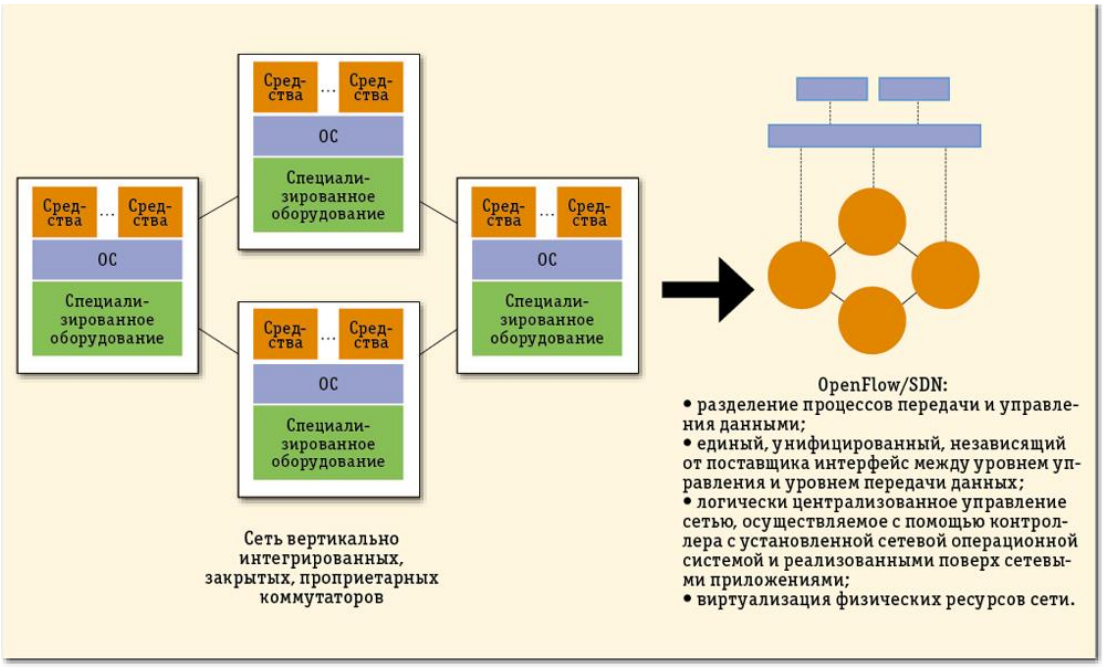

Рис. 1.2. Традиционные сети и ПКС

Согласно спецификации 1.3 стандарта OpenFlow, общение программно-конфигурируемого контроллера с коммутатором осуществляется средствами протокола OpenFlow — в котором любой коммутатор обязан содержать как минимум одну таблицу потоков и групповую таблицу, а также поддерживать канал OpenFlow для связи с удаленным контроллером — являющимся центральным сервером. Спецификация не описывает внутреннее устройство и архитектуру контроллера, а также API для его внутренних приложений. Любая таблица потоков внутри коммутатора должна содержать набор записей с информацией о потоках или правилах маршрутизации. Любая из записей состоит из полей-признаков, счетчиков и набора инструкций. [24]

Механизм работы коммутатора OpenFlow является очень простым. У каждого нового входящего пакета забирается заголовок, который является битовой строкой определенной длины. Далее производится поиск правила в таблицах потоков, для данной битовой маски. В случае нахождения совпадения, над пакетом и его заголовком выполняются различные преобразования, определяемые некоторым набором инструкций, определённых в найденном правиле. Инструкции, ассоциированные с каждой записью таблицы, описывают действия, связанные с отправкой пакета, модификацией его заголовка, обработкой в таблице групп, обработкой в конвейере и отправкой пакета на определенный порт коммутатора. Инструкции конвейера обработки позволяют отправлять пакеты в последующие таблицы для дальнейшей обработки и в виде метаданных передавать информацию между таблицами. Инструкции также определяют правила модификации счетчиков, которые могут быть использованы для сбора разнообразной статистики. [27]

Если нужное правило в не найдено, то пакет инкапсулируется и отправляется контроллеру, тот в свою очередь формирует необходимое для обработки правило. Сформированное правило для определённого типа пакетов загружается в коммутатор или группе подконтрольных коммутаторов.

Также неизвестный пакет может быть сброшен, при установке соответствующей настройки на коммутаторе. [23]

Запись о потоке может сообщать об отправке пакета в определенный физический, зарезервированный или виртуальный порт. [10] Зарезервированные виртуальные порты могут определять общие действия пересылки такие как: отправка контроллеру, широковещательная рассылка, или же пересылка без использования протокола OpenFlow. Виртуальные порты, определенные коммутатором, могут точно определять группы объединения каналов, туннели или интерфейсы с обратной связью. [19]

Записи о потоках могут также хранить информацию о группах, в которых имеется дополнительная обработка. Группы являются набором действий для широковещательной рассылки. Также наборы действий могут представлять из себя пересылки со сложной семантикой, например, агрегирование каналов либо изменение маршрута. Механизм групп позволяет эффективно изменять одинаковые выходные действия для потоков. Таблица групп хранит в себе записи о группах, в свою очередь хранящие список контейнеров действий с особой семантикой, зависящей от типа группы. Действия в одном или нескольких контейнерах действий применяются к пакетам, отправляемым в группу. [8]

Разработчики коммутаторов могут свободно реализовывать любую внутреннюю начинку, однако процедура просмотра пакетов и семантика инструкций должны быть общими для всех. К примеру, в тот момент как поток может использовать все группы для пересылки в некоторое множество портов, разработчик коммутатора может выбрать для реализации этого функционала единую битовую маску внутри большой аппаратной таблицы маршрутизации. Ещё одним примером является процедура просмотра таблиц: конвейер физически может быть реализован с использованием некоторого количества аппаратных таблиц. Установка, обновление и удаление правил в таблицах потоков коммутатора осуществляются только контроллером. Правила могут

устанавливаться реактивно (в ответ на пришедшие пакеты) или проактивно (заранее, до прихода пакетов). [1]

Управление данными в OpenFlow осуществляется на уровне их потоков, а не на уровне отдельных пакетов. Правило в коммутаторе OpenFlow прописывается с участием контроллера только для первого пакета, а все остальные пакеты потока его используют в дальнейшем. [18]

Имеющиеся на сегодняшний день физические коммутаторы ПКС соответствуют пока спецификации OpenFlow 1.0 и содержат только одну таблицу потоков.

1.2.2. Протокол OpenFlow

Идея разработки ПКС заключается в разработке обобщённого и независимого сетевого оборудования. Обладающего единым интерфейсом взаимодействия между передающей средой сети и контроллером. Эта концепция является основополагающей в протоколе OpenFlow. Он позволяет пользователям самостоятельно контролировать и определять, при каких условиях, какие узлы и с каким качеством могут взаимодействовать в Сети между собой. Протокол поддерживает три типа сообщений: контроллер-коммутатор, асинхронные и симметричные. [36]

Сообщения типа контроллер-коммутатор инициируются контроллером и используются для управления и отслеживания состояния коммутатора. Сообщения данного типа могут использоваться контроллером для загрузки конфигурации коммутатора, для выгрузки статистики, для добавления, удаления и модификации записей в таблицах потоков.

Асинхронные сообщения инициируются коммутатором для уведомления контроллера о различных сетевых событиях, например, получение пакетов, удаление записи из таблицы потоков в связи истечения

тайм-аута. Также они могут сообщать об различных изменениях состояния коммутатора и ошибках в результате работы коммутатора. [12]

Симметричные сообщения могут инициироваться коммутатором или контроллером без запроса и используются при установлении соединения. Также используются для измерения пропускной способности соединения контроллер-коммутатор, задержек или для проверки состояния соединения.

1.2.3. Сетевые операционные системы в ПКС

Логически-централизованное управление данными в сети предполагает вынесение извлечение функций управления сетью с передающих устройств на отдельный сервер, называемый программно-конфигурируемым контроллером(ПКК), который находится под контролем администратора сегмента сети. ПКК может управлять одним или несколькими OpenFlow-коммутаторами. Также содержит в себе специальную сетевую операционную систему, предоставляющую сетевые сервисы и унифицированные интерфейсы по низкоуровневому управлению сетью, сегментами сети и мониторинга за состоянием сетевых элементов, а также приложения, осуществляющие высокоуровневое управление сетью и потоками данных. [35]

Сетевая ОС (СОС) является специальной обёрткой над низкоуровневым протоколом. С помощью встроенного API предоставляет приложениям инструменты для доступа к управлению сетью, а выполняет мониторинг конфигурации средств сети. В отличие от традиционного толкования термина ОС, под СОС понимается программная система, обеспечивающая мониторинг, доступ и управление ресурсами всей сети, а не ее конкретного узла. [11]

Подобно классической ОС, СОС обеспечивает программный интерфейс для приложений управления сетью и реализует механизмы управления таблицами коммутаторов: добавление, удаление, модификацию правил и сбор разнообразной статистики. [14] Таким образом, фактически решение задач

управления сетью выполняется с помощью программного обеспечения, разработанного на базе API сетевой операционной системы. Этот инструментарий позволяет создавать приложения на уровне высокоуровневых абстракций, к примеру, имя хоста и имя пользователя, а не низкоуровневых параметров конфигурации, к примеру, MAC- и IP- адресов. Это позволяет исполнять управляющие команды невзирая на базовую топологии сети, однако необходимо, чтобы СОС поддерживала отображения между высокоуровневыми абстракциями и низкоуровневыми конфигурациями.

В каждом контроллере хранится как минимум одно приложение, для управления подключенными коммутаторами. Также программное обеспечение формирует модель подконтрольной топологии физической сети, благодаря этому централизуется управление. [13] Представление топологии сети содержит в себе топологию коммутаторов, расположение хостов и пользователей и других элементов, и различных сервисов сети. Представление также содержит в себе связь между именами и адресами, поэтому одной из важнейших задач, решаемых СОС, является ежесекундный мониторинг сети. Это позволяет СОС создавать приложения в виде централизованного ПО, с использованием высокоуровневых имён, на базе различных алгоритмов, например, алгоритма Дейкстры поиска кратчайшего пути в графе, вместо сложных распределенных алгоритмов вроде алгоритма Беллмана – Форда, в терминах низкоуровневых адресов, которые используются в современных маршрутизаторах. [16]

На данный момент имеется большое количество реализаций сетевых ОС для программно-определяемых сетей. Одними из самых известных на сегодняшний день являются: RUNOS, BigSwitch, Beacon, POX, Maestro, FloodLight, NOX и Trema.

Для контроллеров в ПКС важным требование является то, чтобы все приложения одного контроллера в любой момент времени должны быть синхронизированы и иметь одинаковое представление о топологии сети. Однако переход от распределенного управления сетью к централизованному

таит в себе ряд недостатков. К примеру, снижение надежности, отказоустойчивости и масштабируемости. [1]

Сегодня активно развиваются 3 подход для построения распределенного масштабируемого контроллера. Они называются Kandoo, HyperFlow и Onix. В рамках исследований ЦПИКС было обнаружено, что наиболее перспективным является альтернативный подход, представленный на рисунке Рис. 1.3. В силу того, что любой контроллер может быть соединен с несколькими коммутаторами, а каждый коммутатор может быть соединён несколькими контроллерами, то получается, что есть возможность объединить контроллеры в групповой контроллер (ГК). Группа контроллеров, объединённых в ГК должны иметь синхронизированное представление подконтрольной топологии сегмента сети. Как видно на рис. 1.3, C1 – C3 — контроллеры, S1 – S4 — коммутаторы, а V1 – V3 — фрагменты сети, к которым обеспечивает доступ коммутатор S1, S2, S3 соответственно. Тогда ГК1 образуют контроллеры C1 и C2, ГК2 — C2 и C3, а все приложения в ГК1 должны иметь согласованное представление о топологии V1 и V2, все приложения в ГК2 — о топологии V2 и V3. В случае выхода из строя, например, контроллера C1 его может заменить C2, взяв на себя управление V1. Представление о состоянии соответствующей части сети контроллеры могут согласовывать либо через коммутатор S4, либо через S1, S2 и S3. [31]

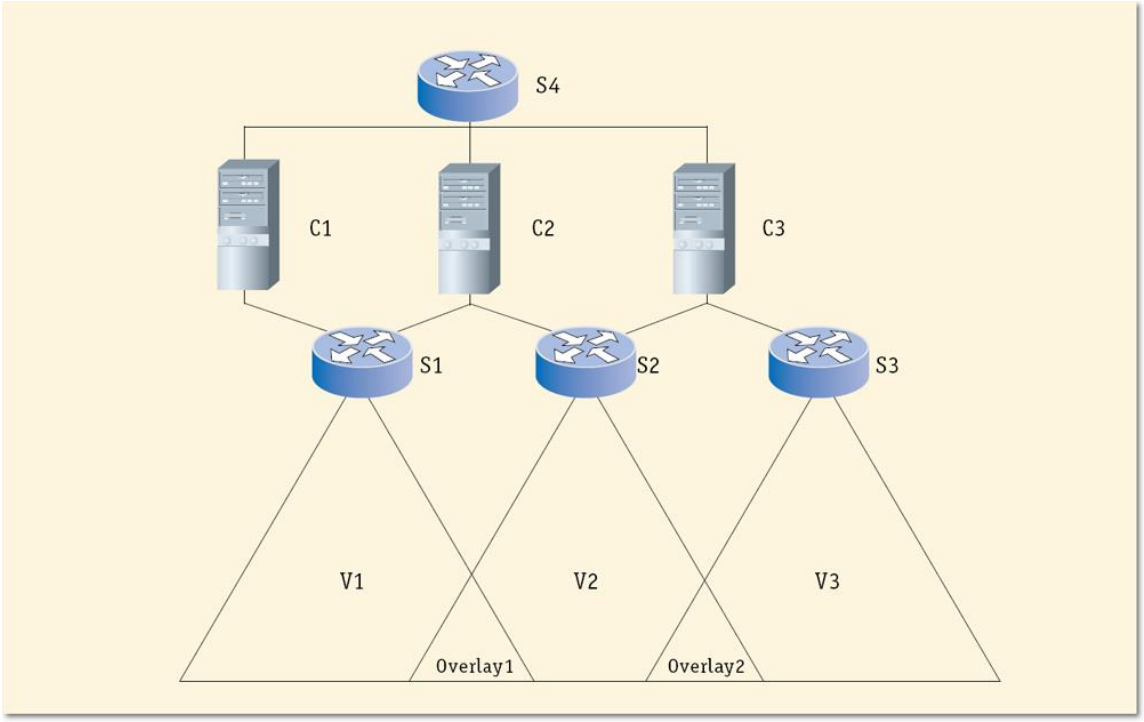

Рис. 1.3. Альтернативный подход к построению распределенного масштабируемого контроллера

Такой принцип построения распределенного контроллера решает проблему масштабируемости и повышает отказоустойчивость ПКС.

1.3. Общие сведения о построении путей

Чаще всего кратчайший путь рассматривается с помощи математического объекта, называемого графом.

Существуют три наиболее эффективных алгоритма нахождения кратчайшего пути:

1. алгоритм Дейкстры (используется для нахождения оптимального маршрута между двумя вершинами);
2. алгоритм Флойда (для нахождения оптимального маршрута между всеми парами вершин);
3. алгоритм Йена (для нахождения k-оптимальных маршрутов между двумя вершинами).

Основной задачей данной научно-исследовательской практики является программная реализация алгоритма поиска кратчайшего пути между двумя любыми вершинами графа. [39]

Программа должна работать так, чтобы пользователь вводил количество вершин и длины рёбер графа, а после обработки этих данных на экран выводился кратчайший путь между двумя заданными вершинами и его длина. Необходимо предусмотреть различные исходы поиска, чтобы программа не выдавала ошибок и работала правильно.

Данная программа может использоваться в дискретной математике для исследования графов или в качестве наглядного пособия, демонстрирующего применение алгоритма Флойда на практике.

Граф — это система, которая интуитивно может быть рассмотрена как множество узлов и множество соединяющих их линий, геометрический способ задания графа представлен на рис. 1.4. Узлы называются вершинами графа, линии со стрелками - дугами, без стрелок - ребрами. Граф, в котором направление линий не выделяется, т.е. все линии являются ребрами, называется неориентированным; граф, в котором направление линий принципиально, т.е. линии являются дугами, называется ориентированным.

Теория графов может также рассматриваться как раздел дискретной математики, а если точнее, то теории множеств, и формальное определение графа является следующим: если задано конечное множество X , состоящее из n элементов (X = {1, 2,..., n}), называемых вершинами графа, и подмножество V декартова произведения, называемое множеством дуг, тогда ориентированным графом G называется совокупность (X , V), неориентированным графом называется совокупность множества X и множества неупорядоченных пар элементов, каждый из которых принадлежит множеству X. Дугу между вершинами i и j будем обозначать (i , j). Число дуг графа будем обозначать m (V = (v1, v2,..., vm)). Пример графа представлена на Рис. 1.4.

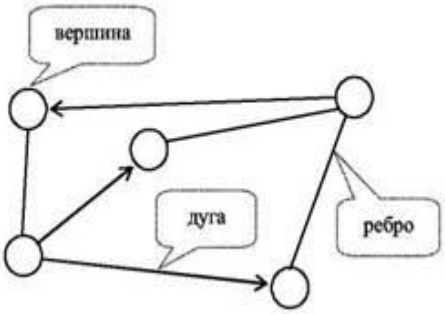

Рис. 1.4. Пример графа.

Язык графов является удобным для описания многих физических, технических, экономических, биологических, социальных и других систем.

В рамках теории графов обычно решаются транспортные и технологические задачи.

- «Транспортные» задачами является огромный спектр логистических задач, в которых вершинами графа являются пункты, а ребрами - дороги или другие транспортные маршруты. Другим примером являются сети снабжения. В них вершинами являются пункты производства и потребления, а ребрами - возможные маршруты перемещения. Соответствующий класс задач оптимизации потоков грузов, размещения пунктов производства и потребления и т.д., иногда называется задачами обеспечения или задачами о размещении. Их подклассом являются задачи о грузоперевозках;
- «Технологические задачи», в которых вершины отражают производственные элементы заводов, станков и т.д., а дугами являются-потоки сырья, материалов и продукции между ними. Суть этих задач заключается в определении оптимальной загрузки всех производственных элементов;

1.4. Выбор инструментальных средств реализации для реализации алгоритма, эмуляции сложного сегмента программно-конфигурируемой сети

1.4.1. Обзор сетевых ОС

С момента развития программно-конфигурируемых сетей, появилось много контроллеров, которые иногда именуют Сетевыми Операционными системами, однако, не все эти контроллеры являются достаточно хорошими. Одними из зарекомендовавших себя являются контроллеры: RUNOS, HP Virtual Application Networks ПКС Controller и POX. Ниже будет произведён подробный обзор контроллеров с целью выявить слабые и сильные стороны каждого. [21]

1.4.1.1. Обзор RUNOS

Начиная с 2013 года, Центр Прикладных Исследований Компьютерных Сетей, начал разработку контроллера RUNOS. По заявлениям ЦПИКС это самый быстрый на сегодняшний момент контроллер по сравнению с существующими аналогами. Данный контроллер очень эффективно использует вычислительные ресурсы для управления сетью. Предлагает огромный набор сетевых приложений, возможность фильтрации трафика, работа с сетевыми протоколами и прочее.

На сегодняшний момент является одной из самых быстрых и эффективных сетевых ОС, за маленькой средней скорости генерации нового соединения – 45 мкс и поддержкой 1 тысячи коммутаторов единовременно. В RUNOS реализована балансировка нагрузки, согласованное видение всей сети, работа с распределёнными сетевыми приложениями, безопасность и противодействие внешним нагрузкам.

Главными характеристиками являются:

- Обработка 30 миллионов потоков в секунду;
- Время на установку нового соединения – 45 мкс;
- Поддержка 1000 коммутаторов;
- Возможность управления из графического интерфейса;

Разработчики заверяют, что на осень 2014 года — это самый быстрый ПКС-контроллер в мире. Его производительность достигается за счет использования возможностей многоядерных и многопроцессорных систем. В нем задействован набор сетевых приложений — как из мира традиционных сетей, так и новых: L2/L3-маршрутизация с учетом качества обслуживания, многопотоковая маршрутизация, фильтрация трафика, работа с сетевыми протоколами (ARP, DNS, DHCP, BGP), трансляция адресов (NAT), балансировка нагрузки, виртуализация сетей, анти-DDoS, верификация сети, интеграция с системой управления ЦОД.

RUNOS в первую очередь ориентирован на корпоративный сегмент. Его потребительской аудиторией в ЦПИКС видят сетевых администраторов и инженеров ЦОД, телеком-операторов, сервис-провайдеров, а также учащихся по направлению «сетевые технологии», исследователей в области компьютерных сетей и разработчиков перспективных сетевых технологий. [31]

1.4.1.2. Обзор контроллера HP Virtual Application Networks SDN Controller

Этот контроллер для программно-конфигурируемых сетей разрабатывается компанией HP. Является платным, однако обладает возможностью работы с несколькими протоколами программно-конфигурируемых сетей.

Компания придерживается своей политики отсутствия привязки программного обеспечения к аппаратным средствам и по этой причине их контроллер обеспечивает возможность взаимодействия с любым сетевым

оборудованием, которое поддерживает протоколы программно-конфигурируемых сетей.

Главными характеристиками являются:

- Максимальное количество поддерживаемых устройств – 4 000;
- Максимальное количество подключенных компьютеров – 30 000;
- Максимальное количество потоков в секунду – 2.3 миллиона;
- Время на установку соединения – 50-60 мкс;

Как видно, данный контроллер является мощным инструментом, однако, время на установку соединение и максимальное количество потоков оставляет желать лучшего. [29]

1.4.1.3. Обзор NOX

NOX был разработан инженерами Nicira Networks параллельно с протоколом OpenFlow. Первоначально был разработан с поддержкой двух языков: C++ и Python. В 2008 г. NOX был опубликован под лицензией GPL и с тех пор этот контроллер является базовым для многих научно-исследовательских групп, которые только приступают к изучению ПКС. NOX ориентирован на дистрибутивы Linux (в частности Ubuntu 11.10 и 12.04, но также возможно использование на Debian и RHEL 6). Содержит сервисы для построения топологии сети и L2-L3 коммутации.

В процессе тестирования выяснилось, что поддержка двух языков сильно сказывается на производительности, поэтому часть, которая отвечает за Python, вытащили в отдельный проект, который позже назвали POX.[26]

1.4.1.4. Обзор POX

Контроллер POX является «младшим братом» NOX. Если при разработке NOX основной целью была высокая производительность, то POX в первую очередь направлен на обучение и исследования. По своей сути POX – это платформа для быстрой разработки и прототипирования ПО управления

сетью. Этот контроллер написан на Python, его легко запустить на Window, Linux и Mac OS. К примеру, исследовательская группа в Стэнфорде использует POX для исследования ключевых проблем ПКС. Является достаточно производительным контроллером, распространяется бесплатно и поддерживает последнюю версию протокола OpenFlow. POX находится в стадии активного развития: все удачные идеи постоянно перемещаются из лабораторных экспериментов в официальные релизы контроллера POX (по крайней мере так утверждают его разработчики из Стэнфорда).

Главными характеристиками являются:

- Максимальное количество устройств – 3000;
- Максимальное количество потоков в секунду – 1 миллион;
- Время на установку соединения – 90 мкс;

POX поддерживает те же компоненты, графический интерфейс, средства визуализации, как и NOX. [28]

1.4.1.5. Обзор Beacon

Это достаточно быстрый, кроссплатформенный, модульный OpenFlow контроллер на Java. Этот контроллер разрабатывается уже более двух лет. Beacon используется во многих научно-исследовательских проектах и тестовых внедрениях/развертываниях. Beacon применяется в экспериментальном ЦОДе Стэнфорда, в котором он управляет 100 виртуальными и 20 физическими коммутаторами

Beacon написан на Java и работает на многих платформах, начиная от высокопроизводительных многоядерных Linux-серверов до смартфонов на Android. Разработчик контроллера — Дэвид Эриксон, ученик Ника МакКеона, Стенфорд.

1.4.1.6. Обзор Maestro

Maestro – это операционная система, разработанная в Rice University. Maestro предоставляет интерфейсы для реализации модульных приложений управления сетью для доступа и изменения состояния сети, а также координации их взаимодействия. Несмотря на то, что этот проект направлен на создание OpenFlow контроллера, Maestro не ограничивается только OpenFlow-сетями. Среда программирования Maestro предоставляет интерфейсы для добавления новых пользовательских компонентов по управлению сетью.

Кроме того, Maestro пытается использовать параллелизм в пределах одной машины для улучшения пропускной способности системы. Разработчики заявляют основными свойствами Maestro портативность и масштабируемость. Maestro разработана на Java (и сама платформа, и ее компоненты), является универсальной для различные ОС и архитектур. [25]

1.4.1.7. Выводы

В результате анализа сетевых ОС был выявлено, что для реализации данной магистерской диссертации лучше всего подходит RUNOS благодаря скорости работы, существенного объёма одновременно поддерживаемых потоков и скорости отклика.

1.4.2. Обзор средств для эмуляции программно-конфигурируемой сети

На текущий момент существует большое количество средств для эмуляции программно-конфигурируемых сетей. Самыми известными и признанными специалистами являются: ns-3 Network Simulator, OPNET, NetSim и Mininet. При выборе средства для эмуляции стоит обратить внимание на функциональные возможности, поддержку необходимых протоколов,

преимущества и недостатки. Также, стоит учесть значимость недостатков эмулятора для поставленных задач.

1.4.2.1. Обзор ns-3 Network Simulator

ns-3 Network Simulator является проектом с открытым кодом, распространяемым по лицензии GNU GPL. Цель которую поставили перед собой разработчики — это создать бесплатный симулятор построения компьютерных сетей, с помощью которого можно производить различные исследования, касательно работы компьютерных сетей, без необходимости строить реальные.

Первая версия ns появилась в далёком 97 году, тогда симулятор был достаточно слабым, несмотря на довольно быстрое ядро, написанное на C++, из-за сценариев, основанных на Tcl, сильно тормозилась работа.

Чуть позже, DARPA, Xerox и ряд других заинтересованных компаний и организаций начали помогать в разработке такого программного обеспечения, спустя несколько лет появился ns-2, который стал более производительным и продвинутым с точки зрения настроек.

Начиная с 2006 года началась разработка третьей версии программы, данная версия имеет существенное преимущество в сравнении со старыми версиями, изменился скриптовый движок, теперь используемые скрипты пишутся на язык Python, который прост в освоении. Помимо этого обеспечена обратная совместимость со второй версии. Релиз третьей версии состоялся в июне 2008 года, после этого проект дорабатывался до очень стабильной версии, в 2010 году, вышла достаточно стабильной, чтобы полностью заменить вторую версию, таким образом в 2010 году, поддержка ns-2 прекратилась. [33]

Самыми главными плюсами данного эмулятора являются: возможность создания топологии, конфигурирования узлов и соединений, анализ нагрузки и визуализация данных.

Самым главным минусом, является отсутствие поддержки протоколов программно-конфигурируемых сетей.

1.4.2.2. Обзор OPNET

OPNET это проприетарное программное обеспечение, разработанное компанией OPNET Technologies. Первая версия программы была представлена публике в 2000 году.

Основная аудитория данного продукта являются коммерческие компании, которым необходимо разработать дата-центр, центр обработки данных или внушительную локальную сеть.

Их инструмент предоставляет удобный интерфейс при работе с эмулированной средой, с помощью такой программы очень просто и быстро строится сеть любой сложности.

Opnet содержит библиотеки, благодаря которым осуществляется формирование телекоммуникационных сетей, и облегчает изучение модели путем подключения различных типов узлов, с использованием различных видов связи и т. д. [32]

Редактор узлов представляет собой редактор, который используется для создания моделей узлов и указания их внутренней структуры. Эти модели используются для создания узлов внутри сети в редакторе проекта.

Внутренние узлы модели имеют модульную структуру, которая определяется как узел подключения нескольких модулей с пакетом потоков и кабелей. Это соединение позволяет обмениваться информацией и пакетами между ними. Каждый модуль имеет определенную функцию в узле, такую как: создание пакетов, склеивание, процесс или передача и прием.

В этом редакторе элементы доступны как черные ящики, корпусу атрибутов, которые могут быть настроены. Каждый из них представляет функцию в узле.

Также имеет мощный редактор модели соединений. Редактор дает возможность создавать новые типы объектов связи. Каждый новый тип соединения может иметь различные атрибуты и представления.

Типы поддерживаемых редактором соединений: все соединения, которые мы можем поддерживать, одно или все четыре, допускаются симулятором. Этими соединениями являются: точка-точка, дуплекс точка-точка, шина и болт. [30]

Объекты представляют в этом редакторе процессоры. Их поведение определяется в процессе редактора. Есть предварительно сконфигурированные модели, такие как источники данных, поглотители и т. д.

Самыми главными плюсами данного эмулятора являются: простота использования, поддержка всех существующих протоколов.

Самыми главным минусом является стоимость данного продукта.

1.4.2.3. Обзор NetSim

Программное обеспечение, разработанное компанией TETCOS. Разработка была начата в июне 2002 года. Это программное обеспечение поддерживает современные протоколы, обеспечивает возможность эмуляции беспроводных сетей. [6]

Проект создан на языке C, что делает симулятор очень быстрым. Данный симулятор является прекрасным инструментом для разработки архитектуры сложных сетей.

В силу своей эффективности и гибкости, данный инструмент является сложным в освоении и используется только профессионалами.

Данный продукт имеет огромную стоимость, что является его главным минусом.

1.4.2.4. Обзор Mininet

Симулятор сети Mininet является бесплатным программным обеспечением разрабатываемым сообществом программистов, которые, являются сторонниками бесплатного программного обеспечения.

Mininet спроектирован таким образом, чтобы можно было легко и просто создавать виртуальные программно-конфигурируемые сети. Помимо этого был разработан способ подключения удалённого контроллера, который позволяет тестировать любые программно-конфигурируемые контроллеры.

Данный эмулятор появился в первую очередь из-за быстро растущего интереса к программно-конфигурируемым сетям. Обладает функциями генерации данных, позволяет тестировать полученную сеть на предмет производительности и сразу выявлять слабые места спроектированной топологии. [9]

Вся эмуляция происходит на уровне ядра. С помощью удобного API, позволяет быстро строить сложные топологии, которые могут хранить не одну тысячу устройств.

1.4.2.5. Выводы

В ходе анализов было выявлено, что Mininet является лидером среди эмуляторов за счёт предоставляемых возможностей и бесплатного распространения.

1.4.3. Обзор сред разработки

С момента развития программирования для упрощения и ускорения разработки были разработаны различные среды разработки, которые предоставляют широкий спектр возможностей и инструментов для ускорения проектирования и написания кода.

За недолгую историю эру программирование было разработано много сред разработки под различные языки программирования. Так как в рамках выполнения проекта нам понадобится язык программирования C++, то мы рассмотрели лучшие среды разработки для языка программирования C++. Лучшими на данный момент являются: Microsoft Visual Studio, Eclipse CDT, NetBeans, Code Lite.

1.4.3.1. Обзор Microsoft Visual Studio

Microsoft Visual Studio – это продукт компании Microsoft разработанный в первую очередь для разработки программного обеспечения под операционную систему Windows. Благодаря модульности имеет поддержку нескольких языков программирования такие как C++, C#, JavaScript и т.д.. Он включает в себя прекрасный редактор исходного кода с поддержкой технологии IntelliSense.

IntelliSense специальная технология автодополнения текста. Как и другие системы автодополнения является удобным инструментом для просмотра описания функций и списков аргументов. С помощью неё производится ускорение разработки программного обеспечения, за счёт уменьшения нагрузки на программиста. Кроме того, она позволяет уменьшить количество обращений к документации, благодаря выводы часть документации в виде всплывающих вспомогательных окон в редакторе кода. В ходе работы IntelliSense производит сканирование метаданных классов, переменных и иных конструкций подключенных библиотек и сохраняет эти данные во внутреннюю базу данных. «Классическая» реализация IntelliSense производит поиск специальных маркеров в коде, например, символ точки. Как только обнаруживается маркер то программисту сразу демонстрируются доступные методы класса, согласно его уровню доступа, а также параметры необходимые для этого метода.

Microsoft Visual Studio является дорогой и качественной средой разработки, однако для студентов и преподавателей может предоставляться бесплатно по специальной программе DreamSpark.

Минуса у этой среды разработки два: сильно ориентирована под разработку под Windows и отсутствие кросс платформенности.

1.4.3.2. Обзор Eclipse CDT

Eclipse это свободная модульная интегрированная среда разработки, написанная на языке Java. История разработки начинается с компании IBM, как приемник IBM VisualAge, в качестве корпоративного стандарта среды разработки, на разных языках под различные платформы IBM. Согласно данным IBM первоначальная разработка стоила 40 миллионов долларов. Исходный код был полностью открыт и сделан доступным после того, как Eclipse был передан для дальнейшего развития независимому от IBM сообществу.

Одним из важнейших преимуществ этой среды разработки является возможность разработки различных расширений, которые могут помочь, программисту в разработке. Уже сейчас имеется мощный инструмент для языка Java под названием Java Development Tools, для C/C++ под названием C/C++ Development Tools. Эти и другие расширения под различные языки были разработаны сообществом совместно с разработчиками IBM.

Благодаря расширениям Eclipse имеет хорошие расширения для работы с различными системами контроля версий, таких как CVS и GIT. Также имеет прекрасные средства для связи с системами управления ошибок типа Youtrack или Jira.

Благодаря бесплатному распространению и высокому качеству программного обеспечения, не сильно уступающему такому тяжеловесу как Microsoft Visual Studio получила большую популярность среди программистов одиночек и огромного количества организаций.

Благодаря тому, что сам продукт написан на Java, среда разработки является кроссплатформенной, что позволяет ему работать на многих платформах, начиная с Windows-подобных систем и заканчивая широким спектром Unix-подобных систем и MacOS.

Архитектура состоит из следующих компонентов:

- Ядро платформы, отвечает за загрузку Eclipse и запуск подключаемых модулей, например, CDT или JDT;
- OSGi;
- SWT;
- JFace;
- Рабочая среда Eclipse Представляющая из себя различные понели, редакторы и т.д.;

OSGi (Open Services Gateway Initiative) это библиотека для динамической модульной системы и сервисной платформы для Java-приложений. Она разрабатывается консорциумом OSGi Alliance. Спецификации дают модель для построения приложения из компонентов, связанных вместе посредством сервисов. Суть заключается в возможности динамической переустановки различных компонентов и составных частей приложения без необходимости в перезапуске.

Круг применений данного стандарта довольно широк. Первоначально разработка велась для создания встроенных систем, но сейчас на базе OSGi строят многофункциональные автономные настольные приложения и корпоративные решения.

OSGi Alliance, ранее известная как Open Services Gateway initiative — организация открытых стандартов (open Standards Development Organization — SDO). В течение последних нескольких лет она разрабатывала основанную на Java сервисную платформу OSGi (также известна как The Dynamic Module System for Java), которая могла управляться удаленно. Основная часть этой

разработки — фреймворк (каркас), который определяет модель жизненного цикла приложения и служебного реестра.

SWT (Standard Widget Toolkit) — специальная открытая библиотека для быстрой разработки графических интерфейсов пользователя на языке Java.

Разработка велась фондом Eclipse. Лицензируется под Eclipse Public License, одной из лицензий открытого программного обеспечения.

SWT не является самостоятельной графической библиотекой, а лишь представляет собой кросс-платформенную оболочку для графических библиотек конкретных платформ, например, под Linux SWT использует библиотеку GTK+. SWT написана на стандартной Java и получает доступ к OS-специфичным библиотекам через Java Native Interface, который рассматривается в качестве сильного средства, несмотря на то, что это не является чистой Java.

SWT является достойной альтернативой для разработчиков взамен AWT и Swing (Sun Microsystems). Он необходим для тех разработчиков, которые желают получить привычный внешний вид программы в данной операционной системе. Использование SWT делает Java-приложение более эффективным и гибким, но снижает независимость от операционной системы и оборудования, требует ручного освобождения ресурсов и в некоторой степени нарушает Sun-концепцию платформы Java.

JFace это большой набор вспомогательных Java-классов, реализующий наиболее общие задачи построения GUI. В рамках проекта Eclipse библиотека JFace имеет следующее описание: «Элементы пользовательского интерфейса, реализация которых может быть утомительной». JFace представляет собой дополнительный программный слой над SWT, реализующий паттерн программирования Model-View-Controller. JFace предоставляет следующие возможности:

1. Предоставляет «Viewer» классы, отвечающие за отображение и реализующие трудоёмкие задачи по заполнению, сортировке, фильтрации, а также обновлению виджетов;

2. Предоставляет «Action» классы, которые позволяют разработчику определять специфическое поведение для отдельных элементов пользовательского интерфейса, таких как пункты меню, кнопки и т. д.;

3. Предоставляет регистры, содержащие шрифты и изображения;

4. Предоставляет набор стандартных диалоговых окон и виджетов, а также предоставляет фреймворк для создания сложного графического интерфейса для взаимодействия с пользователем;

Основная цель JFace заключается в предоставлении разработчику большого количества инструментов для разработки пользовательского интерфейса, позволяя ему в первую очередь сосредоточиться на бизнес-логике приложения.

Основной задачей группы разработчиков Eclipse было сокрытие реализации компонентов графического интерфейса построенных на основе библиотеки SWT и по возможности максимальное использование библиотеки JFace как более высокоуровневой и простой в использовании. Библиотека JFace использует SWT, но SWT является не зависимой от JFace. Однако, рабочая среда Eclipse построена с использованием обеих библиотек и в некоторых местах SWT используется напрямую в обход JFace.

1.4.3.3. Обзор NetBeans

NetBeans IDE — свободная интегрированная среда разработки приложений (IDE) на языках программирования Java, Python, PHP, JavaScript, C, C++, Ада и ряда других.

Проект NetBeans IDE поддерживается и спонсируется компанией Oracle, однако разработка NetBeans ведётся независимым сообществом разработчиков-энтузиастов (NetBeans Community) и компанией NetBeans Org.

Последние версии NetBeans IDE поддерживают рефакторинг, профилирование, выделение синтаксических конструкций цветом,

автодополнение набираемых конструкций на лету и множество предопределённых шаблонов кода.

Для разработки программ в среде NetBeans и для успешной инсталляции и работы самой среды NetBeans должен быть предварительно установлен Sun JDK или J2EE SDK подходящей версии. Среда разработки NetBeans по-умолчанию поддерживала разработку для платформ J2SE и J2EE. Начиная с версии 6.0 NetBeans поддерживает разработку для мобильных платформ J2ME, C++ (только g++) и PHP без установки дополнительных компонентов.

В сентябре 2016 года Oracle передала интегрированную среду разработки NetBeans в руки фонда Apache.

NetBeans IDE поддерживает плагины, позволяя разработчикам расширять возможности среды. Одним из самых популярных плагинов является мощный дизайнер отчётов iReport (основанный на библиотеке JasperReports).

На идеях, технологиях и в значительной части на исходном коде NetBeans IDE базируются предлагаемые фирмой Sun коммерческие интегрированные среды разработки для Java — Sun Java Studio Creator, Sun Java Studio Enterprise и Oracle Solaris Studio (для ведения разработки на C, C++ или Фортран). Сравнительно недавно Sun стала предлагать эти среды разработки бесплатно для зарегистрировавшихся в Sun Developer Network (SDN) разработчиков, сама же регистрация на сайте бесплатна и не требует никаких предварительных условий, кроме согласия с лицензией CDDL.

NetBeans IDE доступна в виде готовых дистрибутивов (прекомпилированных бинарных файлов) для платформ Microsoft Windows, Linux, FreeBSD, Mac OS X, OpenSolaris и Solaris (как для SPARC, так и для x86 — Intel и AMD). Для всех остальных платформ доступна возможность скомпилировать NetBeans самостоятельно из исходных текстов.

1.4.3.4. Обзор Code Lite

CodeLite — свободная кроссплатформенная среда разработки программного обеспечения для языка C/C++ с открытым исходным кодом.

Является простой и незамысловатой средой разработки со встроенной поддержкой интеграции системы контроля версий Subcersion и Git. Имеет автодополнение ctags + clang, удобный рефакторинг кода.

CodeLite распространяется по лицензии GNU General Public License v2 или более поздней версии. Является свободным программным обеспечением. CodeLite в настоящее время, будучи разработан и отлажен, использует себя в качестве платформы разработки.

1.4.3.5. Выводы

Среди представленных выше сред разработки, для разработки эффективного алгоритма оптимизации трафика подходит Eclipse CDT, благодаря своей кроссплатформенности и отсутствия привязки к какой-либо операционной системе.

2. Проектирование и реализация

2.1. Проектирование структуры эффективного алгоритма оптимизации трафика

Суть эффективного алгоритма оптимизации трафика заключается в том, чтобы снизить задержки на создание соединения. В первую очередь стоит обратить внимание на то, что такое потоки в рамках программно-конфигурируемых сетей. Поток данных – это некий маршрут, который прокладывается внутри сегмента сети между коммутатором A и коммутатором B, для того, чтобы n количество пакетов достигла некоторый узел сети.

В программно-конфигурируемом контроллере используется стандартный алгоритм Дейкстры для построения маршрута на графе, что не является совсем быстрым решением, т.к. используется универсальная версия алгоритма с использованием сложных структур данных. В случае его модификации можно произвести улучшение производительности в рамках данной задачи.

Алгоритм Дейкстры не гарантирует нахождения пути на графе, для решения этой проблемы необходимо дополнительно разработать механизм восстановления маршрута из целевой вершины, не попавшей в маршрут.

Таким образом мы получаем, что наш алгоритм будет разбит на две подзадачи первая это разработка модификация алгоритма Дейкстры на графах для ускорения создания новых маршрутов в рамках данной задачи, вторая это разработка быстрого алгоритма восстановления маршрута от конечной вершины, до стартовой в рамках построенного алгоритмом Дейкстры пути. Алгоритм генерации маршрутов всегда должен строить оптимальный маршрут, чтобы исключить ситуацию, необоснованной перегрузки одного коммутатора по отношению к другим, если это возможно.

2.1.1. Проектирование и реализация алгоритма построения маршрутов на базе простых структур данных

RUNOS является контроллером с открытым исходным кодом, а значит мы можем свободно модифицировать исходные файлы. В первую очередь стоит отметить, что изначально использовалась библиотека boost для расчёта маршрута по алгоритму Дейкстры, но имеется ряд минусов, таких как избыточно сложная структура хранения данных и полностью универсальная реализация алгоритма. [37] Это вызвано тем, что библиотека boost является крайне универсальной в своих реализациях, в связи с этим с её помощью не всегда можно получить максимальную производительность. [34]

Первое на что стоит обратить внимание — это структура хранения данных, она является сложной с не самой быстрой скоростью доступа. Классической структурой данных для работы с графами является двумерный массив (Рис. 2.1.). Главной отличительной особенностью является константное время доступа к любой ячейке данных, и оно ничтожно мало, чего не скажешь о сложных структурах данных. Благодаря этому преимуществу массивы позволяют быстро модифицировать связи между отдельными узлами сети. При расчёте маршрута имеется ряд сложностей, и самая главная заключается в том, что для полного обхода всех возможных рёбер одного узла, нам потребуется совершить O(n) операций, где n это количество коммутаторов, а в худшем случае это будет $O(n^2)$.

В большинстве языков программирования массивы проще инициализировать и использовать, нежели сложные структуры данных.

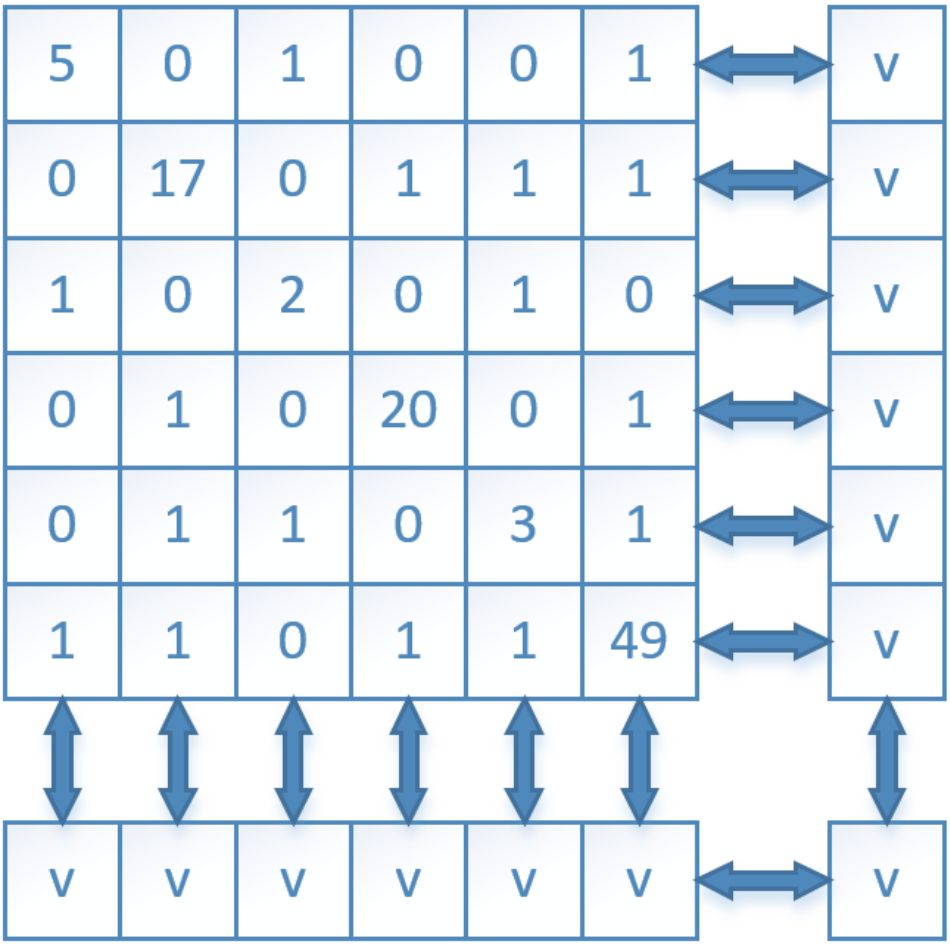

Рис. 2.1. Представление двумерного массива.

Проблему обхода можно будет решить с помощью разработки специальной, простой структуры данных, которая позволит быстро обходить все рёбра определённой точки, благодаря тому, что в ней будут хранится только реальные связи. Помимо преимущества в скорости расчётов, можно имеется одно ещё важное преимущество, это занимаемая память, в большинстве случаев она будет меньше, чем у массива, исключением будет ситуация, когда у каждого коммутатора будет больше 50 соединений с другими устройствами. Важным недостатком такой структуры будет скорость модификации такой структуры.

В листинге 2.1. представлена реализация двумерного массива для хранения данных о топологии сети.

Листинг 2.1. Реализация двумерного массива для хранения данных о топологии сети.

```
graphData = new *int32_t[count];
    for (int32_t i = 0; i < count; i++) {
        graphData[i] = new int32_t[count];
        for (int j = 0; j < count; j++) {
            graphData[i][j] = 0;
        }
    }
```

Для простого взаимодействия с массивом, необходимо разработать обёртку с методами, позволяющими производить различные действия с массивом, такие как инициализация массива, добавление или удаление связи между узлами, увеличение или уменьшение счётчика проходящих потоков. Также внутри обёртки должна быть реализация поиска маршрута в виде data_link_route. Эту логику реализуют следующие методы: addLink, removeLink, incrementFlows, decrimentFlows и computeRoutes.

Метод addLink выполняет генерацию связи между двумя узлами. На листинге 2.2 представлен поиск начального узла в массиве данных.

Листинг 2.2. Поиск начального узла в массиве данных.

```
    map::iterator it = dpidToGraphId.find(from.dpid);
    if (it != dpidToGraphId.end()) {
        fromId = it->second;
    }
    else {
        dpidToGraphId[from.dpid] = switchIdCounter++;
        fromId = dpidToGraphId[from.dpid];
    }
```

На листинге 2.3. представлен поиск конечного узла в массиве данных.

Листинг 2.3. Поиск конечного узла в массиве данных.

```
map::iterator it = dpidToGraphId.find(to.dpid);
if (it != dpidToGraphId.end()) {
        toId= it->second;
}
else {
        dpidToGraphId[to.dpid] = switchIdCounter++;
        toId = dpidToGraphId[to.dpid];
}
```

На листинге 2.4. представлено создание связи между двумя коммутаторами.

Листинг 2.4. Создание связи между коммутаторами.

```
graphData[fromId][toId] = 0;
graphData[toId][fromId] = 0;
```

Метод добавления оказался таким большим в первую очередь из-за того, что из ПКК мы получаем данные в виде структуры switch_and_port, из-за этого нам необходимо сначала найти соответствие между идентификатором и идентификатором в рамках протокола OpenFlow, а после этого произвести установку самого соединения.

Метод removeLink выполняет уничтожение связи между двумя узлами. Перед изменениемНа листинге 2.5 представлено удаление связи между коммутаторами.

Листинг 2.5. Удаление связи между коммутаторами.

```
graphData[fromId][toId] = -1;
graphData[toId][fromId] = -1;
```

Метод уничтожения связи оказался таким большим из-за того, что из ПКК мы получаем данные в виде структуры switch_and_port, в которой

хранится идентификатор в рамках протокола OpenFlow, после нахождения идентификатора в массиве происходит уничтожение связи.

Метод incrementFlows выполняет увеличение счётчика потоков, проходящих через данное соединение коммутатор. На листинге 2.6 представлена реализация метода.

Листинг 2.6. Реализация метода incrementFlows.

```
int fromId, toId;
map::iterator fromIt = dpidToGraphId.find(fromSwitch.dpid);
map::iterator toIt = dpidToGraphId.find(toSwitch.dpid);
if (fromIt != dpidToGraphId.end() && toIt != dpidToGraphId.end()) {
    fromId = fromIt->second;
    toId = toIt->second;
    graphData[fromId][toId]++;
    graphData[toId][fromId]++;
}
```

Метод decrimentFlows выполняет уменьшение счётчика потоков, проходящих через данный коммутатор. На листинге 2.7 представлена реализация метода.

Листинг 2.7. реализация метода decrimentFlows.

```
int fromId, toId;
map::iterator fromIt = dpidToGraphId.find(fromSwitch.dpid);
map::iterator toIt = dpidToGraphId.find(toSwitch.dpid);
if (fromIt != dpidToGraphId.end() && toIt != dpidToGraphId.end()) {
    fromId = fromIt->second;
    toId = toIt->second;
    graphData[fromId][toId]--;
    graphData[toId][fromId]--;
}
```

Непосредственно сам поиск пути разбит на несколько небольших подзадач. Первая подзадача — это построение самого маршрута с помощью алгоритма Дейкстры, в случае нахождения минимального маршрута до интересующего коммутатора завершаем поиск. Второй этап — это восстановление пути из конечной точки, до стартовой.

Для улучшения читаемости кода и объединения повторяющихся операций были разработаны несколько вспомогательных функций, первая это обсчёт минимальных расстояний до определённых узлов имеющая название setMinVals и findNextMinId для поиска следующий точки из, которой необходимо производить следующий поиск.

Метод setMinVals принимает минимальные значения и производит сканирование весов рёбер и производит определение то, насколько далеко находится следующая точка. Реализация метода представлена на листинге 2.8.

Листинг 2.8. Реализация метода setMinVals.

```
int32_t startValue = graphData[fromId][fromId];
for (int i = 0; i < count; i++) {
    int32_t flowsCount = graphData[fromId][i];
    if (flowsCount == -1) {
        continue;
    }
    int32_t currentValue = graphData[i][i];
    if (currentValue == -1 || currentValue > startValue + flowsCount)
{
        graphData[i][i] = startValue + flowsCount;
    }
}
```

Метод findNextMinId производит поиск следующей не посещённого коммутатора, у которого самое минимальное расстояние от стартовой точки. Реализация этого метода представлена в листинге 2.9.

Листинг 2.9. Реализация метода findNextMinId

```
int32_t minVal = -1;
int32_t minId = -1;
for (int32_t i = 0; i < count; i++) {
    if (!visited[i]) {
        currentVal = graphData[i][i];
        if (minVal == -1 || minVal > currentVal) {
            minVal = currentVal;
            minId = i;
        }
    }
}
return minId;
```

Для восстановления маршрута используется специальный алгоритм, реализованный в методе restoreRoute, которому сообщается идентификатор конечного узла и из него алгоритм строит маршрут. Реализация алгоритма представлена в листинге 2.10.

Листинг 2.10. реализация алгоритма восстановления маршрута.

```
data_link_route result;
int32_t prevId = endId;
do {
    result.insert(0, graphIdToSwitch[prevId]);
    prevId = findPrevId();
} while (prevId != -1);
return result;
```

В ходе изучения особенностей работы алгоритма Дейкстры была обнаружена важная закономерность, что из любой конечной точки можно найти маршрут до начальной. Так как в ходе алгоритма каждой точке

производится назначение минимального расстояния от начальной точки, которое является суммой весов всех участвующих в этом соединений, то мы можем увидеть, что если мы будем брать каждое ребро графа и вычитать его вес и проверять значение минимального расстояния, то мы увидим, что это минимальное расстояние совпадает только в том случае, когда мы движемся в сторону нашей начальной позиции. Поиск предыдущей точки реализована в функции findPrevId и представлена в листинге 2.11.

Листинг 2.11. Реализация findPrevId.

```
if (graphData[fromId][fromId] == 0) {
    return -1;
}
for (int32_t i = 0; i < count; i++) {
    if (graphData[fromId][i] < 0) {
        continue;
    }
    if (graphData[i][i] == graphData[fromId][fromId] - graphData[fromId][i]) {
        return i;
    }
}
return -1;
```

Были произведены следующие оптимизации алгоритма Дейкстры, реализованного в ПКК RUNOS:

1. Завершение алгоритма происходит сразу после нахождения минимального маршрута до конечной точки. При генерации случайных соединений, оптимизация позволяет сохранить огромное количество времени, в среднем 10 мс от начального времени;

2. Переход на простую структуру двумерного массива для хранения данных. При операциях построения маршрутов происходит прирост

производительности за счёт гарантированной скорости доступа к каждому ребру и узлу в виде O(1). За счёт этого удаётся выиграть от 10 до 5 мс в зависимости от размера сегмента;

3. Также было оптимизировано хранение уже посещённых вершин в виде массива bool значений. Этот массив создаётся только один раз при старте программы. Такое решение было сделано для ускорения работы, ведь операция генерации и удаления массива является дорогим удовольствием. Изначально этот массив генерировался внутри работы алгоритма, после анализа производительности с помощью профайлера было выявлено, что 2 мс тратится на создание и удаление этого массива и было принято решение создавать его только один раз;

Оптимизированный алгоритм Дейкстры разбит на несколько этапов: инициализация данных, поиск минимальных маршрутов из стартовой позиции, восстановление маршрута. Алгоритм реализован в функции computeRoute. В листинге 2.12 представлена инициализация данных перед поиском.

Листинг 2.12. Инициализация данных перед поиском.

```
for (int i = 0; i < count; i++) {
        visited = false;
        graphData[i][i] = -1;
}
int fromId, toId;
fromId = dpidToGraphId[from.dpid];
toId = dpidToGraphId[to.dpid];
```

После успешной инициализации происходит выполнение первого шага цикла. При первом шаге производится поиск минимальных расстояний из начальной точки до всех доступных. Реализация этого этапа работы алгоритма представлена на листинге 2.13.

Листинг 2.13. Реализация первого шага алгоритма.

```
if (fromId == toId) {
        return route;
}
visited[fromId] = true;
graphData[fromId][fromId] = 0;
setMinVals(fromId);
int32_t nextMinId = findNextMinId(visited);
```

Далее основной цикл повторяет один и тот же шаг, до тех пор, пока не обнаружит построенный маршрут до конечной точки. Как только это происходит, производится выход из алгоритма. Цикл представлен в листинге 2.14.

Листинг 2.14. Основной цикл алгоритма.

```
while (nextMinId != toId) {
        setMinVals(nextMinId);
        visited[nextMinId] = true;
        nextMinId = findNextMinId(nextMinId);
}
```

После построения маршрута выполняется восстановление маршрута в формате понятным для ПКК Runos. Вызов метода представлен на листинге 2.15.

Листинг 2.15. Вызов метода восстановления маршрута.

```
return restoreRoute(nextMinId);
```

Как можно видеть основные шаги сохранились, в первом шаге происходит первоначальный поиск расстояний до других вершин. Следом запускается цикл, в котором производится поиск минимального расстояния до остальных вершин. Как только будет найдено расстояние до конечной

вершины будет произведён выход из цикла с последующим восстанавлением маршрута.

2.1.2. Оптимизация работы алгоритма построения маршрутов на базе простых структур данных с помощью OpenMP

К сожалению алгоритм Дейкстры является рекуррентным алгоритмом и не позволяет произвести вычисление маршрутов параллельно, однако при детальном рассмотрении реализации мы увидим, что данный алгоритм имеет одно место, которое возможно распараллелить. Это место находится в методе setMinVals. В нём происходит простой нерекурентный перебор рёбер и расчёт новых значений минимального маршрута. Реализация параллельного метода представлена на листинге 2.16.

Листинг 2.16. Параллельная версия setMinVals.

```
        int32_t startValue = graphData[fromId][fromId];
#pragma omp parallel for num_threads(4)
        for (int i = omp_get_thread_num(); i < count; i += omp_get_max_threads()) {
                int32_t flowsCount = graphData[fromId][i];
                if (flowsCount == -1) {
                        continue;
                }
                int32_t currentValue = graphData[i][i];
                if (currentValue == -1 || currentValue > startValue + flowsCount) {
                        graphData[i][i] = startValue + flowsCount;
                }
        }
```

Параллелизм в зависимости от сегмента может как ускорить, так и замедлить работу. Для маленьких сегментов сети, в которых меньше 100 коммутаторов, работа замедляется на 5 мс. При работе с крупными сегментами, от 500 до 1000 узлов, наблюдается ускорение производительности работы алгоритма на 3-5 мс.

2.1.3. Эмуляция тестовых сегментов сети

В рамках первой главы были рассмотрены различные инструменты для эмуляции сегментов компьютерных сетей, в ходе анализа было выявлено, что инструмент Mininet является отличным вариантом для тестирования алгоритма построения маршрутов.

Mininet позволяет конфигурировать топологии с помощью скрипта на языке Python. Это позволяет нам генерировать сегменты со случайными связями любой сложности и размерности.

В первую очередь для корректной работы необходимо произвести импорт библиотек для корректной работы средства эмуляции. На листинге 2.17 продемонстрирован импорт необходимых библиотек.

Листинг 2.17. Импорт библиотек.

```
from mininet.net import Mininet
from mininet.node import NOX, OVSSwitch, Controller, RemoteController
from mininet.topo import Topo
from mininet.log import setLogLevel
from mininet.cli import CLI
```

Класс с названием Mininet, находящийся в пространстве имён mininet.net используется для эмуляции сети с логическими компьютерами.

Библиотека mininet.node хранит в себе описание классов, которые отвечают за контроллеры и коммутаторы.

Библиотека mininet.topo используется необходима для корректной работы топологии сегмента сети. В задачи этой библиотеки входит: хранение информации об узлах, создание/удаление/модификация подключений между узлами.

Библиотека mininet.log является вспомогательной библиотекой с помощью, которой производится логирование отладочной информации, вывод тех или иных ошибок. Это очень полезная библиотека на этапе отладки и тестирования, т.к. помогает обнаруживать проблемы практически сразу.

Библиотека mininet.cli необходима для возможности конфигурирования и выполнения различных команд уже после запуска средства для эмулирования, с помощью командной строки.

Создание подключения к удалённому ПКК RUNOS продемонстрированно в листинге 2.18.

Листинг 2.18. Создание подключения к ПКК RUNOS

```
runos = RemoteController('c0', ip='127.0.0.1')
```

Для создания собственной топологии необходимо наследовать класс Topo и после этого можно будет модифицировать его работу. Реализация пользовательской топологии продемонстрировано в листинге 2.19.

Листинг 2.19. Создание пользовательской топологии.

```
class DeikstraTopo ( Topo ):
    def __init__ ( self ):
        Topo.__init__( self )
```

После создания топологии необходимо создать и произвести запуск сети, для этого необходимо добавить все логические коммутаторы в сеть, а после соединить их в случайном порядке, для корректной работы сети. В листинге 2.20 продемонстрирована генерация логических коммутаторов.

Листинг 2.20 Генерация логических коммутаторов.

```
RemoteController = self.addSwitch(‘s0’)
i = 0
while i < topoSize:
        NewSwitch = self.addSwitch(‘deikstraSwitch’ + str(i))
        i = i + 1
        self.addLink( RemoteController, NewSwitch )
i = 0
while i < topoSize:
        Switch = self.getSwitch(‘deikstraSwitch’ + str(i))
        j = 0
        while j < 20:
        ForConnect = self.getSwitch(‘deikstraSwitch’ + str(random.randInt(0,
topoSize))
        self.addLink( Switch, ForConnect)
        j = j + 1
```

После генерации топологии необходимо произвести создание и запуску экземпляра Mininet. Для запуска сети сперва необходимо создать экземпляр топологии DeikstraTopo. Следующим шагом будет передача необходимых параметров для создания экземпляра сети. В конце необходимо будет произвести построение и следом запуск сети. После запуска необходимо вызвать консольную оболочку CLI, которая будет принимать различные команды от пользователя, с помощью которых, во время выполнения можно вносить различные изменения в работу созданной сети.

Создание и запуск представлены на листинге 2.21.

Листинг 2.21. Создание и запуск экземпляра сети Mininet.

```
topo = DeikstraTopo()
net = Mininet( topo=topo, switch=OVSSwitch, build=False )
net.build()
net.start()
CLI( net )
net.stop()
```

Включение Mininet производится с помощью специальной консольной команды, представленной на листинге 2.22. Во входных параметрах мы указываем где находится файл скрипта для запуска и какую топологию использовать.

Листинг 2.22. Команда для запуска Mininet.

```
sudo mn --custom ~/mininet/custom/diplom.py --topo diplom
```

3. Тестирование и апробация

После разработки алгоритма для программно-конфигурируемой сети на базе сетевой операционной системы RUNOS, была произведена эмуляция различных сложных сегментов сети, для определения степени достижения поставленных целей и задач.

В общей сложности было проведено 2 группы тестов. На основе тестов были получены данные о производительности существующего, параллельного реализованного алгоритма и однопоточного алгоритма. Была составлены диаграммы производительности для более понятного отражения результата проделанной работы. На этих диаграммах можно увидеть различие по скорости выполнения построения маршрута на различных размерах сегментов программно-конфигурируемых сетей.

Целью тестов первого этапа было выявить насколько эффективными являются различные реализации алгоритма Дейкстры, на базе различных структур данных и оптимизаций. Полученные данные отражают сильные и слабые стороны того или иного подхода. Схема проведения тестов была различной, первая группа тестов была нацелена чисто на исследование времени генерации соединений. Для исследования времени расчёта маршрута на тестовом сегменте каждые 100 мс создавался запрос на передачу данных из одного случайного коммутатора до другого. Каждый тест данной группы проводился на протяжении 24 часов. Между тестами изменялся только размер сегмента и реализация алгоритма. Результаты тестирования представлены на рис. 3.1.

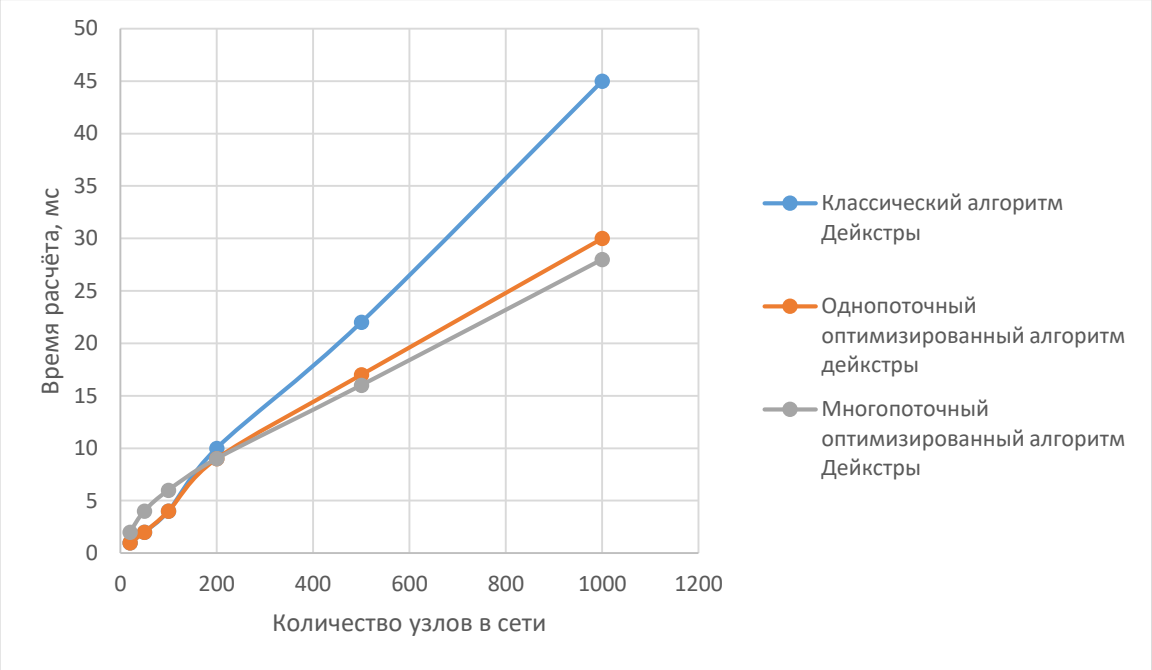

Рис. 3.1. Результат первого этапа тестирования.

Как видно в рамках результата тестирования, при увеличении количества узлов в сети график растёт не линейно для классической реализации и однопоточной оптимизированной реализации. Однако реализация многопоточного оптимизированного алгоритма Дейкстры по времени выполнения немного проигрывает на небольших сегментах сети, из-за увеличенных затрат на создание потоков, для работы в многопоточном режиме. Как можно видеть, на сегменте размеров в 100 узлов, оптимизированный алгоритм Дейкстры работает немного быстрее чем классическая реализация, использованная в сетевой ОС RUNOS. Начиная с 200 узлов, многопоточная версия оптимизированной реализации начинает работать быстрее, т.к. количество обрабатываемых данных увеличивается и вместе с этим растёт и эффективность работы параллельной версии алгоритма.

Второй этап тестов был нацелен на тестирование производительности программно-конфигурируемой сети в условиях передачи маленьких файлов размером 1 – 1024 байта. Результаты второго этапа тестирования представлены на рис.3.2.

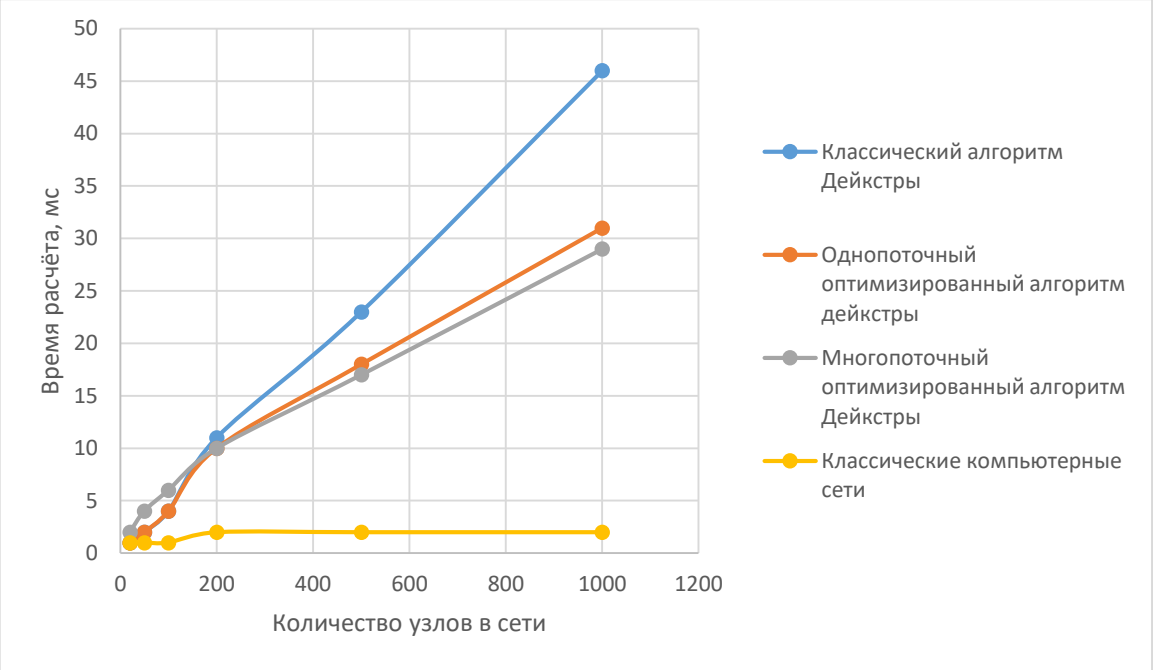

Рис. 3.2. Результаты второго этапа тестирования

Стоит обратить внимание на то, что в рамках данного теста сравнивалась эффективность работы не только между различными реализациями алгоритмов, но ещё и между классическими сетями. Как можно видеть, на маленьких объёмах данных программно-конфигурируемые сети не являются очень эффективным решением. Также стоит заметить, что любой созданный путь в ПКС хранится некоторое время и если происходит повторный запрос на отправку в одно и то же место, то путь не генерируется повторно.

К сожалению, протестировать работоспособность разработанных реализаций алгоритма на реальном сегменте сети не удалось, в связи с тем, что в городе Белгороде отсутствуют программно-конфигурируемые сети.

ЗАКЛЮЧЕНИЕ

Целью выпускной квалификационной работы является исследование алгоритмов построения маршрутов, разработка и оптимизация их скорости выполнения для программно-конфигурируемых сетей. Для достижения указанной цели перед работой был поставлен ряд задач.

При решении задачи «Анализ проблем построения маршрутов в программно-конфигурируемых сетях» в работе была изучена проблема, важность и актуальность данного направления. В ходе изучения было установлено, что такая задача является проблемной и её необходимо решать.

При решении задачи «Выбор инструментальных средств для реализации алгоритма, эмуляции и тестирования реализованного алгоритма на базе программно-конфигурируемых сетей» был произведён подробный анализ различных сред для разработки программного обеспечения, сетевых операционных систем для ПКС, различных систем эмуляции. В результате подробного анализа было выявлено, что для использования в работе подходит среда разработки Eclipse CDT, сетевая ОС RUNOS и эмулятор Mininet.

При решении задачи «Разработка и реализация алгоритма, тестирование на эмулированном сегменте программно-конфигурируемой сети» была произведена разработка оптимизированного алгоритма Дейкстры, с помощью упрощённых структур хранения и усечения лишних шагов алгоритма в рамках ПКС. После этого была произведена настройка эмулятора Mininet и настройка RUNOS для взаимодействия с эмулированным сегментом сети, для отладки и тестирования корректности работы.

При решении задачи «Апробация алгоритма» были произведены исследования работоспособности алгоритма на базе эмулированных сегментов сети различного размера, для выявления преимуществ и недостатков разработанного оптимизированного алгоритма.

Таким образом, задачи были решены полностью, цель была достигнута.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Agouros K., Software Defined Networking / K. Agouros – Берлин: De Gruyter, 2016. – 268 с.

2. Azodolmolky S., Software Defined Networking with OpenFlow – Second Edition / S. Azodolmolky – Бирмингем: Packt Publishing, 2017. – 312 с.

3. Chou E., Mastering Python Networking: Your one stop solution to using Python for network automation, DevOps, and SDN / E. Chou – Бирмингем: Packt Publishing, 2017. – 446 с.

4. Cisco, The Cisco Network Simulator, Router Simulator & Switch Simulator [Электронный ресурс], режим доступа – http://www.boson.com/netsim-cisco-network-simulator, свободный (дата обращения: 17.04.2018);

5. Doherty J., SDN and NFV Simplified: A Visual Guide to Understanding Software Defined Networks and Network Function Virtualization / J. Doherty – Бостон: Addison-Wesley Professional, 2016. – 320 с.

6. Duan Q., Virtualized Software-Defined Networks and Services / Q. Duan, Toy M. – Норвуд: Artech House, 2016. – 334 с.

7. Duan Q., Network as a Service for Next Generation Internet (Telecommunications) / Q. Duan, Wang S. – Стивенидж: The Institution of Engineering and Technology, 2017. – 440 с.

8. Dumka A., Innovations in Software-Defined Networking and Network Functions Virtualization (Advances in Systems Analysis, Software Engineering, and High Performance Computing) / A. Dumka – Херши: IGI Global, 2018. – 364 с.

9. Goransson P., Software Defined Networks, A Comprehensive Approach / P. Goransson – Нью-Йорк: Morgan Kaufmann, 2016 – 436 с.

10. Gray K., Network Function Virtualization / K. Gray – Нью-Йорк: Morgan Kaufmann, 2016. – 270 с.

11. Hamburger V., Building VMware Software-Defined Data Centers / V. Hamburger – Бирмингем: Packt Publishing, 2017. – 432 с.

12. Katti M. Learn About Software-Defined Secure Networks (SDSN) / M. Katti – Саннивейл: Juniper Networks Books, 2016. – 95 с.

13. Khondoker R. SDN and NFV Security: Security Analysis of Software-Defined Networking and Network Function Virtualization (Lecture Notes in Networks and Systems) / R. Knodoker – Берлин: Springer, 2018. – 134 с.

14. Lee G., Cloud Networking: Understanding Cloud-based Data Center Networks / G. Lee – Нью-Йорк: Morgan Kaufmann, 2014. – 238 с.

15. Liyanage M., Software Defined Mobile Networks (SDMN): Beyond LTE Network Architecture (Wiley Series on Communications Networking & Distributed Systems) / M. Liyanage – Хобокен: Wiley, 2015. – 438 с.

16. Mininet Community, Mininet Walkthrough [Электронный ресурс], режим доступа – http://mininet.org/walkthrough/, свободный (дата обращения: 18.12.2017);

17. Morraele P. A., Software Defined Networking: Design and Deployment / P. A. Morraele – Бока Ратон: CRC Press, 2014. – 186 с.

18. Nadeau T. , SDN: Software Defined Networks / T. Nadeau – Себастопол: O'Reilly Media, 2013. – 384 с.

19. NS Community, WHAT IS NS-3 [Электронный ресурс]. режим доступа – https://www.nsnam.org/overview/what-is-ns-3/, свободные (дата обращения: 16.11.2017);

20. OPNET Community, OPNET [Электронный ресурс], режим доступа – https://sandilands.info/sgordon/teaching/resources/opnet.html, свободный (дата обращения: 16.04.2016);

21. Pujolle G. , Software Networks. Virtualization, SDN, 5G, Security / G. Pujolle – Хобокен: John Wiley & Sons Limited, 2015. – 260 с.

22. POX Community, Using the POX SDN controller [Электронный ресурс], режим доступа – http://www.brianlinkletter.com/using-the-pox-sdn-controller/, свободные (дата обращения: 21.04.2018);

23. Qi H., Software Defined Networking Applications in Distributed Datacenters (SpringerBriefs in Electrical and Computer Engineering) / H. Qi – Берлин: Springer, 2016 – 68 с.

24. Robertazzi T.G., Introduction to Computer Networking / T.G. Reobertazzi – Берлин: Springer, 2017. – 154 с.

25. Shukla V., Introduction to Software Defined Networking - OpenFlow & VxLAN / V. Shukla – Сиэтл: CreateSpace Independent Publishing Platform, 2013. – 114 с.

26. Subramanian S., Software Defined Networking (SDN) with OpenStack / S. Subramanian – Бирмингем: Packt Publishing, 2016. – 216 с.

27. Sundararajan R. K., Software Defined Networking (SDN) - a definitive guide / R. K. Sundararajan – Сиэтл: Amazon Digital Services LLC, 2013. – 72 с.

28. Stallings W., Foundations of Modern Networking: SDN, NFV, QoE, IoT, and Cloud / W. Stallings – Бостон: Addison-Wesley Professional, 2015. – 544 с.

29. Taheri J., Big Data and Software Defined Networks (Computing and Networks) / J. Taheri – Стивенидж: The Institution of Engineering and Technology 2018. – 504 с.

30. Tiwari V., SDN and OpenFlow for beginners with hands on labs / Tiwari V. – Сиэтл: Amazon Digital Services LLC, 2013. – 119 с.

31. Zhang Y. Network Function Virtualization: Concepts and Applicability in 5G Networks (Wiley - IEEE) / Y. Zhang – Нью-Йорк: Wiley-IEEE Press, 2017. 179 с.

32. Ермаков А.Е., Основы конфигурирования корпоративных сетей Cisco: учебное пособие / А.Е. Ермаков – Москва: ФГБОУ УМЦ ЖДТ, 2014. – 247 с.

33. Кузьменко Н.Г., Компьютерные сети и сетевые технологии / Н.Г. Кузьменко – Санкт-Петербург: Наука и Техника, 2013. – 368 с.

34. Коннов А.Л., Анализ и проектирование программно-конфигурируемых сетей: учебное пособие / А.Л. Коннов Оренбург: ОГУ, 2016. – 115 с.

35. Максимов Н.В., Компьютерные сети / Н.В. Максимов – Москва: Форум, 2013. – 464 с.

36. Поляк-Брагинский А.В., Локальные сети. Модернизация и поиск неисправностей / А.В. Поляк-Брагинский – Санкт-Петербург: БХВ-Петербург, 2012. – 640 с.

37. Рассел Д., Программно-конфигурируемая сеть / Д. Рассел – Москва: Книга по требованию, 2013. – 96 с.

38. Смирнова Е.В., Построение коммутируемых компьютерных сетей / Е.В. Смирнова – Москва: НОУ ИНТУИТ, 2016. – 429 с.

39. Харари Ф., Теория графов / Ф. Харари – Москва: Ленанд, 2018 – 304 с.

40. ЦПИКС, ПЕРВЫЙ РОССИЙСКИЙ ПКС/SDN- КОНТРОЛЛЕР RUNOS [Электронный ресурс], режим доступа – http://arccn.ru/research/653, свободные (дата обращения: 20.04.2016);

ПРИЛОЖЕНИЕ 1

GraphArray.hh

```
#pragma once

#include "ILinkDiscovery.hh"
#include "exception.hh"
#include "stdint.h"
#include "Topology.hh"
class GraphArray {

    public:
        GraphArray() {
            this->count = 1000;
            switchIdCounter = 0;
            initGraphData();
        };
        ~Graph()
        {
            destroyGraphData();
        }
        void addLink(switch_and_port from, switch_and_port to);
        void removeLink(switch_and_port from, switch_and_port to);
        void incrementFlows(switch_and_port fromSwitch,
switch_and_port toSwitch);
        void decrimentFlows(switch_and_port fromSwitch,
switch_and_port toSwitch);

        data_link_route computeRoute(uint64_t from_dpid, uint64_t
to_dpid);
```

```
    private:
        uint32_t count;
        int32_t **graphData;
        bool *visited;
        int32_t switchIdCounter;
        std::map<uint64_t, uint32_t> dpidToGraphId;
        std::map<uint32_t, uint64_t> graphIdToDpid;
        std::map<uint32_t, switch_and_port> graphIdToSwitch;

        void initGraphData();
        void destroyGraphData();
        void setMinVals(int32_t fromId);
        void setMinVaslParallel(int32_t fromId);
        int32_t findNextMinId(bool *visited);
        data_link_route restoreRoute(int32_t fromId, int32_t endId);
        int32_t findPrevId(int32_t fromId);
};
```

ПРИЛОЖЕНИЕ 2

GraphArray.cc

```
#include "GraphArray.hh"

void GraphArray::addLink(switch_and_port from, switch_and_port to)
{
	int fromId, toId;
	map::iterator it = dpidToGraphId.find(from.dpid);
	if (it != dpidToGraphId.end()) {
		fromId = it->second;
	}
	else {
		dpidToGraphId[from.dpid] = switchIdCounter++;
		fromId = dpidToGraphId[from.dpid];
	}
	map::iterator it = dpidToGraphId.find(to.dpid);
	if (it != dpidToGraphId.end()) {
		toId= it->second;
	}
	else {
		dpidToGraphId[to.dpid] = switchIdCounter++;
		toId = dpidToGraphId[to.dpid];
	}
	graphIdToSwitch[fromId] = from;
	graphIdToSwotch[toId] = to;
	graphData[fromId][toId] = 1;
	graphData[toId][fromId] = 1;
}
```

```
void GraphArray::removeLink(switch_and_port from, switch_and_port to)
{
	int fromId, toId;
	map::iterator it = dpidToGraphId.find(from.dpid);
	if (it != dpidToGraphId.end()) {
		fromId = it->second;
	}
	else {
		dpidToGraphId[from.dpid] = switchIdCounter++;
		fromId = dpidToGraphId[from.dpid];
	}
	map::iterator it = dpidToGraphId.find(to.dpid);
	if (it != dpidToGraphId.end()) {
		toId = it->second;
	}
	else {
		dpidToGraphId[to.dpid] = switchIdCounter++;
		toId = dpidToGraphId[to.dpid];
	}
	graphData[fromId][toId] = -1;
	graphData[toId][fromId] = -1;
}

void GraphArray::incrementFlows(switch_and_port fromSwitch,
switch_and_port toSwitch)
{
	int fromId, toId;
	map::iterator fromIt = dpidToGraphId.find(fromSwitch.dpid);
	map::iterator toIt = dpidToGraphId.find(toSwitch.dpid);
	if (fromIt != dpidToGraphId.end() && toIt != dpidToGraphId.end()) {
```

```
                fromId = fromIt->second;
                toId = toIt->second;
                graphData[fromId][toId]++;
                graphData[toId][fromId]++;
        }
    }

    void GraphArray::decrimentFlows(switch_and_port fromSwitch,
switch_and_port toSwitch)
    {
        int fromId, toId;
        map::iterator fromIt = dpidToGraphId.find(fromSwitch.dpid);
        map::iterator toIt = dpidToGraphId.find(toSwitch.dpid);
        if (fromIt != dpidToGraphId.end() && toIt != dpidToGraphId.end()) {
            fromId = fromIt->second;
            toId = toIt->second;
            graphData[fromId][toId]--;
            graphData[toId][fromId]--;
        }
    }

    data_link_route GraphArray::computeRoute(uint64_t from_dpid, uint64_t
to_dpid)
    {
        for (int i = 0; i < count; i++) {
            visited = false;
            graphData[i][i] = -1;
        }
        int fromId, toId;
        fromId = dpidToGraphId[from.dpid];
```

```
        toId = dpidToGraphId[to.dpid];
        if (fromId == toId) {
                return route;
        }
        visited[fromId] = true;
        graphData[fromId][fromId] = 0;
        setMinVals(fromId);
        int32_t nextMinId = findNextMinId(visited);
        while (nextMinId != toId) {
                setMinVals(nextMinId);
                visited[nextMinId] = true;
                nextMinId = findNextMinId(nextMinId);
        }
        return restoreRoute(nextMinId);
}



void GraphArray::initGraphData()
{
        graphData = new *int32_t[count];
        for (int32_t i = 0; i < count; i++) {
                graphData[i] = new int32_t[count];
                for (int j = 0; j < count; j++) {
                        graphData[i][j] = -1;
                }
        }
        visited = new bool[count];
}
```

```
void GraphArray::destroyGraphData()
{
	for (int i = 0; i < count; i++) {
		delete[] graphData[i];
	}
	delete[] graphData;
	delete[] visited;
}


void GraphArray::setMinVals(int32_t fromId)
{
	int32_t startValue = graphData[fromId][fromId];
	for (int i = 0; i < count; i++) {
		int32_t flowsCount = graphData[fromId][i];
		if (flowsCount == -1) {
			continue;
		}
		int32_t currentValue = graphData[i][i];
		if (currentValue == -1 || currentValue > startValue + flowsCount)
{
			graphData[i][i] = startValue + flowsCount;
		}
	}
}

void GraphArray::setMinVaslParallel(int32_t fromId)
{
	int32_t startValue = graphData[fromId][fromId];
#pragma omp parallel for num_threads(4)
```

```
            for (int i = omp_get_thread_num(); i < count; i += omp_get_max_threads()) {
                int32_t flowsCount = graphData[fromId][i];
                if (flowsCount == -1) {
                    continue;
                }
                int32_t currentValue = graphData[i][i];
                if (currentValue == -1 || currentValue > startValue + flowsCount) {
                    graphData[i][i] = startValue + flowsCount;
                }
            }
        }

        int32_t GraphArray::findNextMinId(bool * visited)
        {
            int32_t minVal = -1;
            int32_t minId = -1;
            for (int32_t i = 0; i < count; i++) {
                if (!visited[i]) {
                    currentVal = graphData[i][i];
                    if (minVal == -1 || minVal > currentVal) {
                        minVal = currentVal;
                        minId = i;
                    }
                }
            }
            return minId;
        }
```

```
data_link_route GraphArray::restoreRoute(int32_t fromId, int32_t endId)
{
	data_link_route result;
	int32_t prevId = endId;
	do {
		result.insert(0, graphIdToSwitch[prevId]);
		prevId = findPrevId();
	} while (prevId != -1);
	return result;
}

int32_t GraphArray::findPrevId(int32_t fromId)
{
	if (graphData[fromId][fromId] == 0) {
		return -1;
	}
	for (int32_t i = 0; i < count; i++) {
		if (graphData[fromId][i] < 0) {
			continue;
		}
		if (graphData[i][i] == graphData[fromId][fromId] -
graphData[fromId][i]) {
			return i;
		}
	}
	return -1;
}
```